ISI/SR-76-5
May 1976

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Validation of Critical Conditions

Jim Carlstedt

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/Marina del Rey/California 90291
(213) 822-1511

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER ISI/SR-76-5	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Protection Errors in Operating Systems: Validation of Critical Conditions		5. TYPE OF REPORT & PERIOD COVERED Research Rept.,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jim Carlstedt		8. CONTRACT OR GRANT NUMBER(s) DAHC15-72-C0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order #2223 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, VA 22209		12. REPORT DATE May 1976 13. NUMBER OF PAGES 34
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* ----		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This document is approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

---

18. SUPPLEMENTARY NOTES

---

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Computer security, operating system security, program verification, protection evaluation, software security

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

See (OVER) NEXT Page for abstract

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20.

ABSTRACT

This report describes a class of operating system protection errors known as "insufficient validation of critical conditions," or simply "validation errors," and outlines a scheme for finding them. This class of errors is recognized as a very broad one, lying outside the scope of the basic protection mechanisms of existing systems; the extent of the problem is illustrated by a set of validation errors taken from current systems. Considerations for validity conditions and their attachment to variables and to various types of control points in procedures are explored, and categories of validation methods noted. The notion of criticality itself is analyzed, and criteria suggested for determining which variables and control points are most critical in the protection sense. Because a search for validation errors can involve substantial information processing, the report references existing or developing tools and techniques applicable to this task.

ISI/SR-76-5
May 1976

ARPA ORDER NO. 2223

PROTECTION ERRORS IN OPERATING SYSTEMS:

Validation of Critical Conditions

Jim Carlstedt

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223 PROGRAM CODE NO. 3D30 AND 3P10.

CONTENTS

ABSTRACT

This report describes a class of operating system protection errors known as "insufficient validation of critical conditions," or simply "validation errors," and outlines a scheme for finding them. This class of errors is recognized as a very broad one, lying outside the scope of the basic protection mechanisms of existing systems; the extent of the problem is illustrated by a set of validation errors taken from current systems. Considerations for validity conditions and their attachment to variables and to various types of control points in procedures are explored, and categories of validation methods noted. The notion of criticality itself is analyzed, and criteria suggested for determining which variables and control points are most critical in the protection sense. Because a search for validation errors can involve substantial information processing, the report references existing or developing tools and techniques applicable to this task.

This work has been performed under Advanced Research Projects Agency Contract DAHC15 72 C 0308. It is part of a larger effort to provide securable operating systems in DOD environments.

ACKNOWLEDGMENTS

Many of the ideas in this report were clarified, and their presentation improved, as a result of suggestions by my collaborators on the Protection Analysis Project, Richard Bisbey and Dennis Hollingworth.

1. INTRODUCTION

This document is one in a series of related reports, each of which describes a specific class of protection errors found in current computer operating systems and presents techniques for finding errors of that type in a variety of systems (different versions, manufacturers, etc.). These reports are intended primarily for protection "evaluators," persons responsible for improving the security of existing operating system software by finding protection errors (fixing them is regarded as a separate maintenance function), and secondarily for designers and students of operating systems. These studies, suggested by the pattern-directed methodology proposed in [Carlstedt75], are intended to assist individuals having no particular expertise in the field of operating system security to effectively carry out the evaluation task, i.e., to find existing errors. However, an evaluator is expected to possess a good working knowledge of the target system, including an understanding of the basic protection mechanisms of the supporting machine.

As used in these studies, the term "protection evaluation" denotes searches for errors based only on static information, primarily program listings but possibly other documentation as well. The ultimate purpose of protection evaluation is to reduce security losses due to protection policy violations made possible by errors in operating system code. The immediate purpose is to detect those errors. The static methods of evaluation discussed in these reports are intended to complement dynamic methods, i.e., system testing, auditing, penetration attempts, etc.

The primary purpose of this report is to assist evaluators to attain the following goals:

1. To gain a better understanding of validation policies and mechanisms and therefore ways validation errors can occur.

2. To obtain the policy information needed for evaluation of a particular target system.

3. To carry out the evaluation itself more methodically and hence more effectively.

The report is organized as follows. In Section 2, the subject of validation is introduced informally via several examples of validation errors taken from current operating systems. In Sections 3, 5, and 7, validation is described more analytically: in Section 3 in terms of its position in the general protection policy-mechanism framework; in Section 5 in terms of the attachment, to variables and control points, of validity conditions determined from considerations of the intended meanings of variables and the requirements of procedures; and in Section 7 by identifying categories of validation methods. Section 4 presents a static view of an operating system as a set of intercommunicating procedures, and suggests that information regarding procedure-variable input-output relationships can be usefully extracted from the target system in preparation for a general evaluation. In Section 6 criteria are suggested for determining which variables and control points are most "critical." Section 8 outlines an evaluation scheme that employs techniques similar to those of program verification.

In those sections of the report where specific tasks are described, an attempt is made to distinguish those activities most amenable to automation from those best done manually and to point out applicable tools and techniques. Some of the tools needed to completely automate a search for validation errors are not yet available. In view of the difficulty of performing some of the required

manual activities, an exhaustive search for validation errors in a large operating system would currently require a very large effort. Nevertheless, in many situations the expected payoff is sufficient to make at least a limited search worthwhile, using a combination of available manual and automatic techniques. On the whole, however, this report should be regarded more as an analysis of what is involved in a search for validation errors than as a prescription for an actual search. It is hoped that this report, in addition to its primary purpose, might also encourage the development of evaluation tools and the design of future operating systems having greater reliability and evaluability.

2. MOTIVATION FOR THE STUDY

The virtual or underlying machine--including both hardware and software "kernel" levels--on which any current operating system is based provides some set of access control mechanisms concerned with preventing certain types of operations except under specified conditions. However, the protection capabilities of such mechanisms are usually quite limited. First, basic access control mechanisms are usually capable only of protecting aggregations such as files, segments, or storage areas. Second, the conditions enforced are usually limited to a predefined set, e.g., possession of a numeric key, ownership by a given user, or membership in predefined categories. Third, enforcement usually occurs only when the operators of a certain predefined set are invoked--"accesses" such as "read," "write," and "execute."

Within an operating system, however, it is frequently necessary to specify conditions that are nonstandard, or that must be enforced at points other than standard access points, or that must apply to objects at a finer level of aggregation. The activity of checking for or insuring such conditions, which must then be specified as part of the operating system software itself rather than as part of the underlying machine, is called "validation." In its explicit form, validation is carried out by means of code for this purpose embedded in procedures at or prior to (in the control sense) the points where such conditions are to hold.

Validation has many uses. One of the most common is to enforce conditions on parameters passed to system procedures by user programs, often to protect against malfunction of those procedures. Less commonly, validation is applied to output to user programs from system procedures. Of more interest here, because the implications of errors are generally more subtle, are validations performed on variables internal to the operating system, and/or those carried out by system procedures other than those at the user interface. An important reason for internal validation is to protect the system against the effects of errors in the system itself. Internal validation of user-specifiable or -influenceable variables is also sometimes necessary when such variables serve as inputs to later operations. Finally, because parts of the operating system may have privileges that exempt them from some of the basic protection contraints, they must sometimes include compensating validation code.

The following examples not only illustrate the importance of validation in various situations, but also indicate the variety of conditions enforced. Most of these examples are paraphrased or adapted from descriptions of protection errors detected in actual operating systems, and communicated to the Protection Analysis Project by various individuals or groups.

1. A system procedure used a user-supplied parameter k as an index to a table of addresses of other system procedures, where k was not validated with respect to the subscript range of the table:

```
                    ...
                    BRANCH   jumptable(k)
                    ...
                    ...
jumptable           BRANCH   procedure-1
                    BRANCH   procedure-2
                    ...
                    BRANCH   procedure-n
                    ...
```

By supplying the appropriate value the invoking procedure could cause the system procedure to execute any instruction in its address space. In this case the system procedure executed in a privileged state and its address space included all of main memory, so that the user could cause any procedure of his choosing (including his own) to be executed in that privileged state.

2. A system file renaming procedure failed to validate the "owner" property of the file to be renamed, allowing the user to replace the name of a system procedure with the name of one of his own.

3. A utility program for copying data from one location to another, intended to be used only by system programmers for low-level maintenance operations but indirectly invokable by user procedures, failed to check that the source location was within the caller's domain of readability (or that the destination was within the caller's domain of writeability).

4. A system procedure providing a certain type of file access required as a parameter a pointer to one of a set of system-private file descriptors representing files "opened" by that user. The validation to insure that the object pointed to was a bona fide file descriptor belonging to that user consisted in part of determining that a particular field contained a pointer to a system-private user account record. However, it was sometimes possible for a user to discover the location of his account record, and hence to provide a pointer to a fabricated descriptor for any file in the system.

5. A block of contiguous storage was used for containing outstanding user requests for a certain serially sharable service. The system controlled the frequency of requests from any one process, to prevent undue delays in servicing requests from other processes. The program managing that service did not check for possible overflow of this table, whose length was defined and made "adequate" at system initialization. A user program could cause the table to overflow, and thus destroy certain system information, by creating a large number of processes, each repetitively requesting the given service.

6. A device allocation procedure A saved the caller's allocation request until the requested device became available. Included in the allocation request was the address of a location x into which the availability notice was to be stored. The address was validated at the time of the request to insure that the caller was authorized to write into x. Immediately after the request was made, the caller could request deallocation of the storage area containing x, and invoke another system service B that was known to require allocation of that amount of storage for certain internal data. Depending on the activity of other processes, there was some probability that the storage allocated to B would be the same as that deallocated from the caller, and containing x. When the originally-requested device became available and the availability notice was stored in x, internal data of B would then be destroyed.

7. One of the first fields of the descriptor of an executable module contained its length, so that storage could be allocated in advance of loading it for execution. Other fields contained the lengths of various components together with the base offsets of locations into which they were to be loaded. The loader failed to validate that these were consistent with the specified overall length, properly assuming that the compiler or linker that produced these modules specified correct component lengths and offsets. However, since these load modules were not protected from

modification by their owners, a user could change offset specifications, causing portions of the module to be placed in unallocated storage, replacing whatever happened to reside there at the time.

8. An input/output buffering procedure B, requiring as a parameter a pointer f to a file descriptor, was callable both by user programs and by other system procedures. When called by a user program, it correctly validated f to insure that it designated a file to which access was authorized. When called by a system procedure, it did not validate f, under the assumption that access was authorized. One of the system procedures that invoked B was a user-callable procedure A for reading records selected by name from indexed files. Procedure A also required a parameter of the same type as f, which it passed directly to B under the assumption that B validated user-supplied parameters. (This is commonly known as the "passthru" error, and is one of of types of validation problems mentioned in [McPhee74].)

As is typically the case with program errors, most of the above appear foolish in retrospect, and most of them could have been prevented in any of a number of ways. The error in example (1), for example, could have been prevented by simple type-enforcement of the sort defined in many programming languages and performed during compilation or provided by compiler-generated code; many of the other errors would not have occurred in systems with access control mechanisms based on adequate domain restrictions. All of the above errors resulted, however, from some form of insufficient validation, due to incorrect design or implementation.

The above examples do not illustrate all varieties of validation. From them and other observations, however, we know that conditions to be validated can involve composite or abstract objects; that they can involve properties of variables other than simple data value; that they can involve relationships among components of a variable or relationships with other variables; that validation can be made more complex by the fact that it is sometimes distributed rather than localized; that mismatches can occur among the validations performed by several procedures; that it may not be readily apparent whether or to what degree a particular variable is critical in the sense that an error in its validation results in possible security violations. In summary, it is apparent that validation is a pervasive and easily misunderstood phenomenon.

3. VALIDATION AS A BRANCH OF PROTECTION

This section presents a brief overview of protection, and shows how validation relates to other areas.

In the context of protection in operating systems, the terms "policy" and "mechanism" have been used to distinguish designer- and user- produced specifications from hardware and software elements that implement or enforce them [Jones73, Levin75]. The term "policy" will be used in this report to refer only to design specifications. In this sense, protection policy is the reference information by which errors in protection mechanisms are defined or identified.

Protection policy exists at various levels of abstractness or detail. At the most abstract level, it can be stated and classified in terms of the user interests it is aimed at protecting. Usually, the term "protection" is associated with "information," and protection policy is aimed at protecting various user interests associated with information privacy and integrity. Another frequently included interest is system reliability, in particular users' interests in not having their service disrupted unexpectedly [Saltzer75]. An interest not usually included in the protection category is that of system efficiency.

Less abstractly, an element of protection policy specifies two things:

1. A condition that is intended to hold at certain times during the operation of the system

2. The times at which it is to hold.

The time component must denote some set of intervals or points in time, such as "whenever control resides in any file manipulation procedure," "whenever more than j outstanding requests exist for service S," "at all points of access of variable x," or "whenever operator W is invoked." The condition component may specify any condition whatsoever, such as "process P must have a privilege level of at least i," "the value of x must be less than the sum of those in table y," or "interrupts must be disabled."

Protection policy is made even more concrete by specifying the condition component in terms of variables defined in the system and the time component in terms of actual system operations. Variables involved in a protection condition must themselves be valid, so that the condition will be accurately represented. Thus they must themselves be the objects of protection conditions. Policy that specifies conditions that are to hold over intervals must be enforced as policy that is to hold at points, i.e., at the occurrence of certain operations, since current computers do not normally provide the capability for continuous checking or enforcement of conditions. The necessity to enforce interval conditions by enforcing point conditions leads to class of protection errors called "consistency errors" [Bisbey75].

As described thus far, protection is a general concept that includes validation as well as such common notions as "exception handling" and "access control." Validation and access control are concerned with preventing or inhibiting certain operations unless certain conditions are met, for example by returning error indications to invoking procedures or aborting current processes, as

opposed to exception handling, where the specified operations may be continued or retried [Goodenough75]. Validation (in its explicit form) differs from access control in that it is not restricted to the underlying machine level and therefore addresses itself to a broader class of conditions and operators. Access control is oriented toward preventing the invocation of "kernel" operators (which need not be primitive) unless certain conditions hold at the time of invocation. Validation may be applied to any operator, it may occur at points other than the invocation of specified operators, and the class of conditions enforced is limited only by programming feasibility.

4. TARGET SYSTEM NORMALIZATION

This section presents the view of a target system appropriate to protection evaluation--that of a set of data-connected procedures and variables. It discusses a "normalization" of the target system, to display these connections for later use, as a possible first phase of the overall evaluation process.

4.1 TARGET SYSTEM DEFINITION AND IDENTIFICATION

The target system consists of a set of procedures distinguished by some degree of mutual trustworthiness and by a particular set of access privileges. (Thus this discussion applies to any subsystem operating within a common protection "domain" [Lampson69].) All other procedures are called "user" procedures.

It is important that all target procedures are identified. This set can contain several hundred elements for a large system such as OS/360 or Multics. To overlook one or more system procedures is potentially to ignore one or more information paths over which validation may be deficient. It is especially important that no outer (user-callable) procedures are missing, since their absences cannot be methodically detected. Given all outer procedures, together with those procedures that receive control as the result of events of predefined types (including interrupts), a list of all system procedures can be generated via a program that recognizes call constructs in the implementation language(s).

4.2 SYSTEM COMMUNICATION GRAPH

The task of searching for validation errors requires knowledge of data flow relationships among procedures and variables. The term "variable" is used here in the sense of a physical or virtual storage cell, possibly structured of other variables, that contains a value or values (possibly null). Variables are not to be confused with identifiers (or selectors in general), by which they are denoted in procedures; the same variable may be denoted by different identifiers in different procedures (or other units of identifier scope), and different variables by the same identifier in different procedures. A frequent problem of static evaluation is that of determining the actual variable accessed via a given identifier at a given control point in a procedure, a problem discussed briefly below.

The variables of primary interest are "nonlocal" or "shared" variables, those accessed by more than one procedure and thus serving as interfaces between procedures. Three types of access-sharing by a given procedure A must be recognized:

1. Via formal parameters of entry points to A.
2. Via actual parameters in calls by A to other procedures.
3. Via non-parameters, or "globals."

These are not mutually exclusive: an actual parameter may also be a formal parameter or a global. To assist in the definition of a data path from one variable to another, it is convenient to

regard the occurrence of an identifier as an actual parameter, as distinct from its occurrence as a formal parameter or global. It is therefore assumed that every occurrence of a formal parameter or global x as an actual parameter is replaced by the identifier x' of a local variable whose value is identical to the value of the variable denoted by x.

It is also assumed that the variable denoted by any identifier in A can be distinguished as either an input variable of A, or an output variable of A, or both. An actual parameter denotes an input (output) variable of A if and only if the corresponding formal parameter of the called procedure denotes an output (input) variable of that procedure.

If the value of an output variable y of A can be influenced by the value of an input variable x of A as a result of processing specified by A, a "local data path" exists from x to y. Local data paths from input to output variables of A can be represented by a bipartite directed graph called a "local data flow graph" (Figure 1).

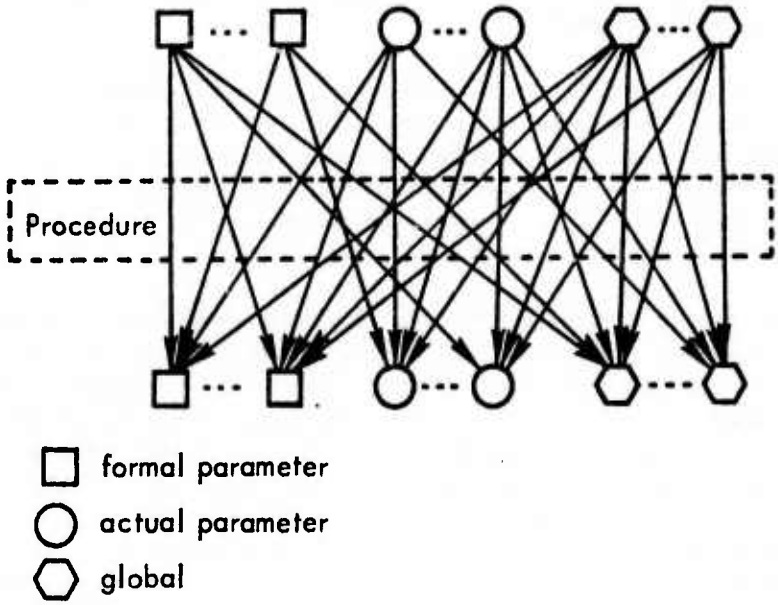

Figure 1. Local data flow graph

The set of local data flow graphs for the procedures of a system, connected in a certain way, is called the system "communication graph." Each of the input (output) nodes of the local data flow graph of a procedure A is connected, via one or more directed arcs, to one or more output (input) nodes of the local data flow graphs of other procedures. Nodes representing formal parameters of A are connected to nodes representing corresponding actual parameters in calls to A ; nodes representing actual parameters are connected to the corresponding formal parameters of the called procedure; and nodes representing globals are connected to nodes representing (global identifiers denoting) the same variable. A "data path" from one variable or procedure to another now has the usual graph-theoretic definition, as do the notions of "antecedent" and "subsequent." The concept of a data path is distinct from that of a "control path," defined as a sequence of directed arcs in the control graph of the system (consisting of flow charts of each of its procedures, with calls

represented by arcs to entry points of called procedures). Both concepts are used in subsequent sections of this report.

4.3 PRODUCTION OF THE COMMUNICATION GRAPH

The task of searching for validation errors is greatly facilitated by having an actual representation of the communication graph on hand. Production of such a representation is part of a class of protection evaluation tasks known as "normalization"--extracting from the target system the information relevant to the evaluation and representing it in a form convenient for the evaluation proper. As noted in [Carlstedt75], a tradeoff generally exists between the amount of work involved in normalization and the amount of work involved in the evaluation. As also noted, normalization can generally either be done completely as a distinct preliminary phase, or it can be done piecewise during the evaluation on an as-needed basis. If the search is to be exhaustive, so that most of the nodes in the communication graph would likely be involved, it is probably more efficient to produce it completely at the outset. Hereafter, the term "communication graph" is used to denote the actual representation as well as the concept.

The first step in producing the communication graph is to recognize the identifiers of variables declared and used within each target procedure. This requires a capability to parse programs written in the given implementation language--a capability that already exists as part of any compiler for that language. The set of input and output access points in a given target procedure must also be determined. This requires recognition of entry points and call constructs, as well as occurrences of identifiers in global accesses.

The second step is to determine the influencing relationships between the input and output variables of each procedure. This task is called "data dependency analysis." To some extent it can be done automatically, with a program that recognizes constructs in which the value of a variable is influenced by the values of one or more others. Dependencies among variables denoted by the actual parameters of a call statement cannot be determined until the dependency analysis for the called procedure has been performed, implying a constraint on the order in which the communication graph can be constructed. An experimental data dependency analysis program is described in [Bisbey76]. In that program, "influenced by" relations are produced for each such construct, the transitive closure of which yields the "influenced by" relations among the outputs and inputs of the procedure as a whole. This indicates a potential influenceability, independent of possible control flow in the procedure. A more precise method, based on control flow analysis, is given in [Allen76]. Neither of these methods can detect subtle influences of the types discussed in [Lampson73] or [Denning75]. Also, there do not (and cannot) exist completely general static data flow analysis programs that can deal satisfactorily with such phenomena as addressing functions (e.g., index computations), program-manipulable bindings (e.g., the use of reference variables), dynamically structured variables, and variables that overlap in physical storage.

The final step in the production of the communication graph is to identify the connections between the input and output nodes of the data flow graphs for individual procedures. For formal and actual parameters the task is relatively straightforward [Bisbey76]. For static bindings to simple global variables, construction of the mapping from local identifiers to globally-uniquely-identified variables can be automated; this is a task performed by loaders and linkage editors. However, the variables to which a procedure has potential read or write access do not necessarily correspond to the identifiers declared in it. Reference variables, for example, may

represent whole sets of potentially accessible variables; a declared identifier, on the other hand, may denote an array or structure of which only a portion can be potentially accessed. The identification of variables actually accessed involves the same set of problems as those encountered in data dependency analysis mentioned above. For implementation languages that allow generalized address calculations and manipulation of bindings, complete automation of this task is not possible.

5. *VALIDATION POLICY*

The second phase of the overall search process is that of gathering the reference information necessary to determine where to search for and how to judge potential errors. This section describes the basic structure of this information as it is formulated by designers and evaluators and as used in the evaluation process, and discusses the major considerations that determine its content.

5.1 VALIDITY CONDITIONS AND CRITICAL ITEMS

An element of validation policy consists of a "validity condition," a logical predicate Cv(Z) expressed in terms of variables Z and "attached" to a control point v (an arc in the flow chart of a procedure). To say "Cv(Z) is attached to v" is simply to say that the assertion "Cv(Z) must be true whenever control reaches v" has been postulated or deduced by designers or evaluators. Assertions of this type do not ordinarily have explicit representations in programming languages or operating system code. If postulated by designers, such an assertion may be represented in design documentation, possibly in the form of a comment in a program listing. However, the evaluator must be open to possibilities for inaccuracies in such documentation, and must endeavor to formulate Cv(Z) as the "actual" required condition at V, as determined also by his/her knowledge of system requirements and/or by inspection and analysis of system itself, particularly that section of procedure subsequent to v (in the uninterpreted flow chart).

The output of this phase of the search process is a set of "critical items," each of the form [v,Cv(Z)]; v is called a "critical point" and Cv(Z) a "critical condition." The critical item serves as the basis of a unit of evaluation activity called a "section evaluation," outlined in Section 8, where a section is that portion of the containing procedure consisting of control paths leading to v. Every control point is a candidate for a critical point; the cost of evaluation requires that only certain control points be selected. Control points vary greatly in their "criticality," i.e., in the consequences of insufficient validation and thus in their need for evaluation. Criteria for selecting critical points are suggested in Section 6.

5.2 INPUT AND OUTPUT CONDITIONS

The critical points on which evaluations are based do not coincide in general with the points of most interest to designers. The major points to which validity conditions are attached by designers are various points of access by procedures to their input and output variables, since these conditions constitute the specifications for the procedure interfaces. The common phrase "validity (or validation) of a variable" is meaningful only in the context of input and output. A validity condition attached to an input (output) point w and specifying domain (range) of validity of the variable x accessed at that point, is called an "input (output) condition" on x, and is denoted by Cw(x,Z). The variable x, an element of Z, is the "object" of Cw(x,Z); the other members of Z are the "reference-state variables."

Validity conditions are not attached to all read and write accesses to shared variables. Rather, the input and output points chosen typically represent sequences of read and write accesses associated with single identifiable purposes. For example, a sequence of read accesses to copy the value of a variable to several other variables might be represented by a single input point. Similarly, a sequence of read and write accesses whose combined purpose is to effect a single overall change in the data value or other properties of a (possibly structured) variable might be

represented by a single output point. However, a number of input and/or output points to the same variable can exist within the same procedure. With the exception of read accesses intermediate to validation sequences, read or write accesses for which different validity conditions apply must be distinguished as separate input or output points, respectively. The points most often (but not necessarily) regarded as input and output points for formal and actual parameters are entry points and call points, respectively.

An "interface error" is a design error that occurs when input and output conditions on the same variable are inconsistent, usually as a result of having been specified independently (i.e., by different designers and/or at different times). Interface errors almost always result in validation errors (although not necessarily, since design specifications might not be obeyed by implementations). When an interface error is encountered, the question arises as to which condition is wrong--the input or the output--or whether either can be regarded as the fiducial policy against which the other must be judged to be in error. Such questions are more properly the concern of system designers and maintainers. The task of the evaluator is to find validation errors, not to classify them in terms of causes.

Because of the possibility of overlooking interface errors, in general output points are not the best choices for critical points. If an output point were chosen as a critical point, the corresponding evaluation, which includes only control paths antecedent to that point, would not detect a validation error resulting from an interface error between an output condition attached to that point and an input condition on the same variable. Input points are also in general not the best choices for critical points, but for a different reason, discussed in Section 7.2.

5.3 FUNCTIONAL VALIDITY VERSUS INTEGRITY

Input and output conditions are determined by two major considerations or factors:

1. Procedure functionality.

2. Variable integrity.

A functional condition on a variable x is determined by the requirements of one or more procedures to which x serves as input. More precisely, it is determined by the requirements of "using sections" corresponding to input points of x. The using section corresponding to an input point u consists of that portion of the containing procedure A made up of control paths starting at u and terminating at other input points of x (to which different functional factors apply), or at exit points of A. A functional factor consists of the specification of the necessary and sufficient conditions on x in order for one or more particular using sections to function correctly.

An integrity condition on a variable x is determined by considerations of the intended meaning of x itself, independent of considerations of the functional requirements of particular procedures that use it. Integrity conditions are especially applicable to variables defined in advance of the procedures that use them, and intended to represent objects, phenomena, or situations of the "real-world" (e.g., the time of day) or of the operating system itself (e.g., the state of some process or resource) as "data base" variables. An integrity condition C(x,Z) defines for x a range (or domain) of validity by specifying relations that the value or other properties of x (such as type,

structure, size, and values of protection attributes) must satisfy with the reference-state variables of Z. The following are examples: "*time_of_day* lies in the range [0:0:0.0, 23:59:59.9]"; "*request_queue* contains no more than m entries"; and "dispatching priority of every class C process is higher than any of those of class D." A variable that satisfies (fails to satisfy) its integrity condition is "inherently valid (invalid)." (A variable may become invalid in a number of ways, e.g., because of the misbehavior of a procedure that modifies it, failure to update it in a timely way, or access by an unintended procedure.) The integrity of a variable is vital to the proper functioning of the system as a whole, as opposed that of particular procedures. It is of obvious importance, for example, for a variable serving as a reference-state variable in an access control specification, where an invalidity can lead to a wrong protection decision.

Normally the meaning of a variable, and hence its integrity condition, does not change over its lifetime. A variable whose meaning varies represents "actual" storage being shared by "virtual" variables.

The distinction between integrity and functional validity corresponds to the duality of concern inherent in the software system design and programming tasks--concern with choice of data representation versus choice of algorithm. Both types of factors may be included by designers in an output condition Cw(x,Z), whetner consciously or not: functional factors on the basis of the assumed requirements of one or more sections for which x serves as input, and integrity factors on the basis of the assumed meaning of x. An input condition Cu(x,Z) normally coincides with the functional condition on x determined more locally by the requirements of the using section corresponding to u. A possibility for interface errors is inherent in the distinction. Both types of factors must be considered by evaluators in determining critical conditions.

6. CRITICALITY CRITERIA

This section addresses the problem of selecting, from among the thousands of candidate control points in a typical target system, those for which the validity conditions are most "critical" and for which the validation is thus most in need of evaluation. This is an economic necessity for an evaluation project of reasonable magnitude. An imprecise and incomplete set of criticality criteria are suggested. To some extent, this task is interdependent with that of determining validity conditions themselves: criticality is better understood when validity conditions have been formulat d, but the formulation of validity conditions assumes knowledge of system structure and requirements, and hence of the real and potential effects of violations of those conditions, which is the basis for estimating criticality.

6.1 THE CHICKEN-AND-EGG VIEW

The term "critical" applies not only to validity conditions attached to control points, but is also (more commonly) used as an attribute of procedures and variables. The criticality of a control point follows from the criticality of the procedure in which it is contained. The criticality of a procedure is judged at least partly by the criticality of its output variables, whose values might become invalid as a result of its misbehavior. The criticality of a variable, in turn, can be judged to some extent by the criticality of the procedures accessing it as input.

To avoid a purely circular definition, some notion of "fundamental" criticality must be introduced, as is done below. Meanwhile, we note that since the criticality of procedures and their input/output variables are so closely related, we need not continually include both types of objects in the discussion. In the remainder of this section we focus on the criticality of variables, with the understanding that the real goal of the evaluator is to identify a set of most critical control points.

6.2 FUNDAMENTAL CRITICALITY

The criticality of a variable is ultimately a measure of the seriousness of the effects of an invalid value of that variable, defined in terms of potential effects on the users of the system. It is a measure of costs of potential violations of high-level protection policy--policy expressed in terms of user expectations for information privacy and integrity and for system capacity and reliability. (Unfortunately, actual costs defined in terms of these ultimate criteria can rarely be estimated with any degree of precision.)

It follows that "fundamentally critical" variables are those closest to the "user interface," i.e., for which invalid values have the most immediate effect on user programs or on protection interests in general. The interface consists of those variables whose values are directly readable by user programs, such as those returned as arguments of user-program calls to system procedures as well as those elements of the system data base directly readable by user programs. These can be readily enumerated. Certain other variables whose values impinge directly on user interests, such as those involved in scheduling or access control decisions, must also be regarded as being included in the user interface.

Fundamental criticality is a measure of the directness as well as the cost of the effects of an invalid value. A relevant question is "What would happen if the value of this variable were such-and-such?" If the effects on user interests are not apparent, the variable is probably not

fundamentally critical (although it might be highly critical in one of the indirect senses discussed below). Note that just determining whether "such-and-such" values are valid or not is equivalent to formulating a validity condition for the given variable.

6.3 INFLUENTIALITY

The potential cost of an invalidity is not necessarily proportional to the directness of the effect. A serious invalidity in a variable at the user interface might be caused by a previous invalidity in a variable removed by several levels of procedure. Thus the actual measure of criticality is the overall influentiality of a variable on user interests, not the directness of the influence. For "internal" or fundamentally noncritical variables this is a two-step phenomenon: their influentiality on values of fundamentally critical variables, and the influentiality of the latter on user interests. These must be measured or estimated differently; the term "influentiality" is used in two distinct senses.

Influentiality in the nonfundamental sense can be estimated in a gross fashion by extending the data dependency analysis techniques mentioned in Section 4.3 to the interprocedural level. This is conceptually similar to data dependency analysis at the intraprocedural level. The program reported in [Bisbey76] performs both intraprocedural and interprocedural analyses for variables shared and accessed as parameters. "Influenced by" relations among inputs to and outputs from any set of procedures are calculated by generating the transitive closure of these relations holding between the inputs and outputs of the individual procedures. A more general algorithm for interprocedural data dependency analysis is described in [Allen74]. Most of the work involved is the same as that required to generate the communication graph itself, as discussed in Section 4.3.

Knowing that a data dependency exists between a given internal variable and a given fundamentally critical variable is not sufficient; what is required is some measure of how strongly the former influences the latter. This suggests a numeric approach, with the criticality of a variable calculated as an "averaging" function of the criticalities of the most immediately subsequent (influenced) variables, weighted to reflect the "strengths" of the functional relationships represented by the procedures connecting them. In Figure 2, for example, the criticality c2 of variable x2 might be calculated as the sum of w1*c0, w2*c0, and w3*c1. In view of the existence of cycles in the system communication graph, such a calculation must proceed iteratively, and the averaging function must be chosen to guarantee convergence to a limiting value of criticality for each variable. This approach is similar to that of calculating information flow in networks [Ford62], with the w's representing arc capacities.

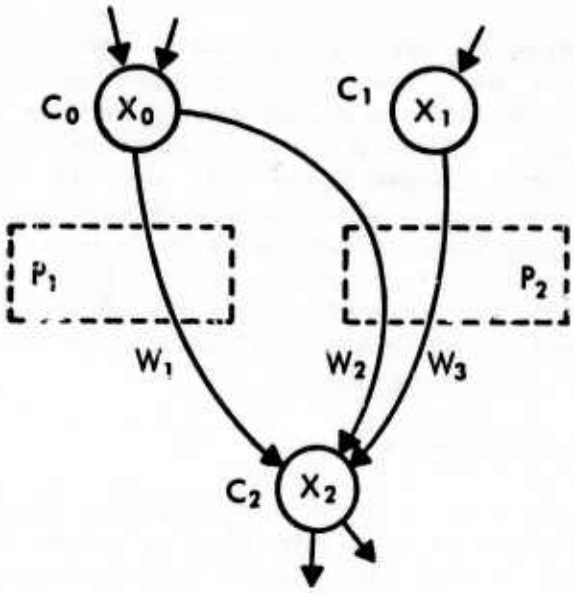

Figure 2. Input-output relationships represented by dependency weights

Such an approach is only a heuristic device to help identify more critical or eliminate less critical variables, since the weights assigned to the arcs cannot capture the algorithmic relationships between input and output variables.

6.4 INFLUENCEABILITY

The "cost" of an invalidity is not just the cost of the effects of a single occurrence, but rather the overall probable costs of whatever invalidities might occur from time to time in the value of a given variable. Thus, included in an estimate of its criticality must be an estimate of the frequency of such invalidities.

Invalidities are of two categories:

1. Those resulting from functional errors in the system.

2. Those due to the successful efforts of malicious users.

The frequency of invalidities of the first type is commonly referred to as "system unreliability" (in the functional sense). Reliability data in terms of occurrences of various types of errors (described symptomatically) is maintained as a standard operating practice by many large installations, although not usually with reference to the variables affected.

The frequency or likelihood of an invalidity resulting from the efforts of malicious users depends (in addition to the usually unknown level of such efforts) on the ease with it can be achieved. This is the "influenceability" of the variable. Influenceability is the exact dual of

influentiality, and can be determined or calculated in a similar manner. Variables whose values depend only on user-supplied inputs are more strongly influenceable than those dependent also on the values of other variables not under control of a single user, e.g., variables representing resource allocation states and total demands. (Some variables are presumably totally noninfluenceable, namely those that reflect only user-independent system status, e.g., total resources.) At the same time, even though a variable might be strongly influenceable in the above sense, the strength of the functional dependence on user-modifiable variables might be masked by several levels of indirectness, i.e., it might be derived only from variables that are derived from variables, and so forth, that are derived from variables that are directly derived from user-supplied inputs. Similar to the distinction between fundamental and nonfundamental influentiality, the apparent influenceability of a variable is not the same as its actual influenceability.

6.5 INCOMPLETENESS OF CRITICALITY CRITERIA

No closed set of criteria can serve as a completely satisfactory and dependable basis for estimating criticality. Control points or variables more critical by any obvious set of criteria are also likely to have received more attention during system development, so that errors in their validation are less likely. At the same time, more subtly critical objects might be more attractive for purposes of intentional exploitation, since errors in their validation might be considered less likely to have been detected and their exploitation less likely to be observed.

7. *VALIDATION MECHANISMS AND THEIR SPECIFICATION*

The final phase of the overall evaluation process consists of carrying out the specific evaluations based on given critical items. These can sometimes be performed more efficiently if they are guided by knowledge of major validation methods in general, or knowledge of design specifications regarding validation methods for the target system in particular. This section discusses the major categories of validation methods and their specification, and the effect of such specifications on an evaluation.

7.1 ENFORCEMENT SPECIFICATIONS

At a more concrete level, an element of validation policy consists of an "enforcement specification," a logical predicate Ev(Z) attached by designers to a control point v, specifying a condition intended to be checked via a routine for that purpose inserted at v. (Also specified is the action to be taken if Ev(Z) does not hold, preventing progress past v without some intervening corrective action; this is not of concern here.) Such checking is "explicit" validation. If Ev(Z) implies the validity condition Cv(Z) attached to v it is "complete;" if Ev(Z) implies only one or more of the major conjunctive terms of Cv(Z) it specifies a "partial" validation. If Ev(Z) is neither complete nor partial it is irrelevant, and a policy error exists. Comments similar to those of Section 5.2 regarding interface errors apply here as well.

Strictly speaking, the evaluator requires no knowledge of validation enforcement specifications for the target system, nor of validation methods employed in operating systems in general. In principle, an evaluation is more honest without such information, since it is less likely to be biased by expectations of finding certain validations performed in certain places. (An exception to the above is the need for an evaluator to be aware of the protection mechanisms of the underlying virtual or physical machine--mechanisms that insure certain classes of conditions at certain points or over certain periods. For the most part, such mechanisms in current systems are concerned with enforcing domain isolation, i.e., preventing cross-domain accesses, but other built-in checking, for example that of data types and lengths on "tagged architecture" machines, may also be relevant.) On the other hand, knowledge of enforcement specifications and awareness of validation methods can make an evaluation more effective by suggesting where attention should be most heavily focused. If, for example, Ev(Z) is known and is complete with respect to a critical condition Cv(Z), then an evaluation with respect to Cv(Z) need include only the validation specified by Ev(Z). Although its insufficiency does not prove that a validation error exists, it does indicate a strong probability. Further evaluation with respect to Cv(Z) is unwarranted. Instead, the implementation of Ev(Z) should be corrected.

Hereafter, when speaking of an actual explicit validation, we shall use Ev(Z) to denote its description rather than its specification, i.e., to denote the actual condition checked.

7.2 EXPLICIT INPUT AND OUTPUT VALIDATION

To show that Cv(Z) is insured by a complete validation or by a set of partial validations, it is necessary to show that

1. The conditions validated logically imply Cv(Z).

2. The validations cannot be skirted.

3. The elements of Z are themselves valid.

Because of the advantages of localization on the evaluation task, three categories of localized explicit validation deserve to be noted. The most obvious method of enforcing $Cv(Z)$ is to implement a complete validation $Ev(Z)$ at v. This is essentially a software-implemented access control routine for the section of procedure subsequent to v. It can be evaluated by comparing $Ev(Z)$ with $Cv(Z)$ directly.

The next most immediate possibility is the distribution of a set of complete or partial validations $\{Ev'(x,Z)\}$ within the containing procedure at points v' on control paths terminating at v and starting at input points of variables x whose values influence the values of elements of Z at v. This is called "input validation." $\{Ev'(x,Z)\}$ is sufficient to insure $Cv(Z)$ if and only if $Cv(Z)$ is implied by the conjunction of the conditions validated along each such path, and item 3 in the above list holds.

Continuing the discussion started in Section 5.2 regarding the distinction between the critical points identified by evaluators and the access points of most concern to designers, it is evident that in general input points are not the best choices for critical points. Where input validation exists, critical points must be chosen as those points (frequently only one) at which the input validation has been completed, so that it will be included in the scope of the evaluations based on those critical points.

Input validation must be employed in situations where elements of x cannot be trusted to be valid, for example where they are supplied by user-written procedures. It is also frequently the most efficient method, in particular for situations where x is modified by a relatively large number of procedures and used by a relatively few, as in the case of a procedure called by many others. Where the same validity condition $C(x,Z)$ must be validated via input validations in several procedures, these validations are sometimes centralized in the form of a single identical routine (e.g., a macro) or a shared procedure. If $C(x,Z)$ applies to *all* uses of x, validation can be centralized even more strongly in the form of a single "use-manager" procedure embodying a validation that is complete with respect to $C(x,Z)$.

The third method deserving special note is output validation. Explicit validation with respect to an output condition $Cw(x,Z)$ can take the form of a complete validation at w itself. A set of partial validations distributed along control paths leading to w cannot be evaluated in the same way as those constituting an input validation, since w represents the point at which the computation of x is completed; implicit validation is involved, as defined below.

Output validation is efficient when $C(x,Z)$ applies as a common factor in the validity conditions attached to a relatively large number of input points and when x is modified by a relatively small number of procedures. An important advantage of output validation is that it represents ***prevention*** of an invalidity, as opposed to input validation, which represents ***detection.*** Output validation allows sources of invalidities to be more easily identified. As was the case with input validation, a distinct validation routine can be associated with each output point of x, or common validation code can be centralized in a single routine. Again, for a common validity

condition C(x,Z), output validation can be centralized into a single mandatory "modification-manager" procedure. The encapsulation of one or more variables behind a set of manager procedures, with an entry or "gate" for each type of access allowed, is a well-known method of insuring information privacy and integrity [Schroeder72].

Validation with respect to a condition C(x,Z) can of course be shared among partial input and prior partial output validations Eu(x,Z) and Ew(x,Z) respectively, where

$$[\ Eu(x,Z)\ \text{AND}\ Ew(x,Z)\ ] \Rightarrow C(x,Z).$$

A partial or complete validation of Cv(Z) that occurs at an antecedent control point v' is relevant only if no modifications to the variables involved can occur during the time that control passes from v' to v. In other words, the consistency of Z must be maintained during that interval. While possible invalidities in the elements of Z due to insufficient validation are included in the domain of "validation errors" relative to Cv(Z), possible invalidities due to modifications by concurrent processes are not. As noted in Section 3, such possibilities represent a distinct class of protection errors.

7.3 GENERALIZED VALIDATION

Up to this point, validation has been assumed to occur in the form of routines that evaluate given conditions and prohibit or inhibit processing if their values are false. A search for validation errors that limits itself to the analysis of explicit validations might result in the detection of apparently insufficient validation in cases where validity is actually assured. Validation of Cv(Z) occurs not only in the explicit or "constraintive" form, but also in the implicit or "progressive" form. It includes *all* program elements on paths to v that contribute to insuring that Cv(Z) is satisfied, computational as well as conditional. If the procedure containing v is such that Cv(Z) will be satisfied whenever a condition Cu(Z) is satisfied at an antecedent control point u, then Cv(Z) is "validated with respect to Cu(Z)" on paths from u to v. Ultimately, validation in this general sense is merely another way of describing what computer programs do.

Let an output condition Cw(y,Z) be validated with respect to an input condition Cu(x,Z), where w and u are an output and input point in procedure A. Cu(x,Z) may be insured by validation of x prior to u, i.e., the validity of y as an output of A is determined by validation of outputs of procedures antecedent to A. Similar statements might be true of the outputs of those procedures, and so on. In the search for validation errors relative to Cw(y,Z), it might be necessary to include a number of procedures antecedent to A before any conclusive results can be obtained.

Undue reliance by designers on implicit validation, specifically on the assumed correctness of implemented procedures and the validity of their inputs, may result in protection errors. It is easy to underestimate the likelihood of contaminated input or of lingering implementation errors. Explicit validation, even if considered redundant, is the best way to insure validity. To paraphrase a programming maxim, "One should include a large amount of explicit validation in a program until it has been checked out, and then leave it there."

8. SUFFICIENCY EVALUATION

This section describes a scheme for the unit of evaluation based on a given critical item, i.e., a scheme for finding errors in validation relative to a given validity condition attached to a given control point.

8.1 OVERALL SCHEME

The input to the evaluation proper is the set of critical items of the form [v,Cv(Z)] discussed in Section 5. The activity described below is the evaluation of that part of the validation relative to Cv(Z) that occurs within the procedure containing v. This evaluation is based on the analysis of actual procedure code, starting at v and working in a direction opposite that of control flow. It differs in this respect from program verification in general, where the object is to show the consistency or inconsistency of conditions attached to various points connected by control paths, and where the analysis can in general proceed in either direction.

The evaluation within the procedure containing v may not establish the sufficiency or insufficiency of the entire validation of Cv(Z) in the target system. If not, its output, as described below, will be a set of additional critical items involving control points in antecedent procedures and requiring the same type of local evaluation activity. Thus the overall evaluation process is recursive. Activities triggered by separate critical items may proceed in parallel.

8.2 SECTION EVALUATION: DERIVATION OF CONDITIONS

Rather than examining the entire containing procedure A, the evaluation of the validation relative to Cv(Z) need only consider that section S consisting of control paths that lead to v in A, and that start at input points of variables that influence Z. When S is small relative to A, and especially when manual analysis is required (which is currently the case), S may for convenience be isolated in advance and the rest of A "scissored away." This can be done visually, by first partitioning A into "basic blocks" and then identifying the basic blocks of S by recursively applying the rule that a basic block is in S if there is a conditional branch from it to a basic block in S, where S initially consists only of the basic block containing v. Routines for this purpose exist in programs for control flow analysis and program optimization [Allen70, Kildall73]. Identification of basic blocks in S is also required as part of the condition derivation process described below.

A basic block can be regarded as having the form [F(X),Q(X)], where F specifies a sequence of basic operations on the set X of variables accessed by A, and has a single entry point, and Q is a control construct involving a (possibly null) set of predicates q1(X), q2(X), . . . , qn(X) corresponding to branches to n distinct basic blocks, possibly including the given block (branch k is taken if and only if qk(X) is true). (See Figure 3.) Basic operations range from simple assignment statements to procedure calls. Q is represented in programming languages by some form of the generalized "case" statement, of which "if-then-else" statements and "go to" statements are instances.

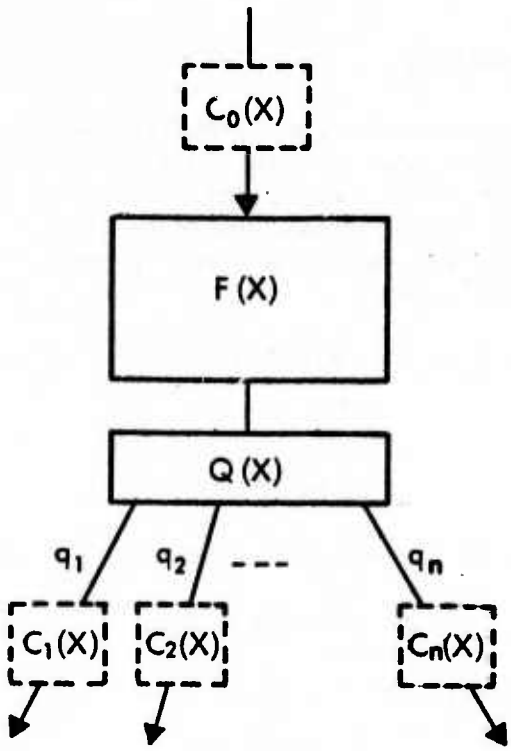

Figure 3. A basic block with associated validity conditions

The evaluation of S is a process of derivation of validity conditions, starting with Cv(X) at v (note the slight change in the interpretation of the argument of Cv) and proceeding along control paths leading to v, a basic block at a time, in the reverse direction of control flow, until input conditions have been "finally" determined for every input point of section S. When the starting validity condition for a basic block B has been derived, it is attached to every branch leading to B from any basic block in S.

If in this manner the condition Ck(X) is attached to branch k of B, then the factor

$$qk(F(X)) \Rightarrow Ck(F(X)) \qquad (1)$$

must be included as a conjunctive term in the expression for C0(X), the starting condition for B. Eventually, validity conditions will be attached to exactly those branches leading from B to other basic blocks in S (conditions for the other branches are irrelevant to Cv(X)). If, for example, these are the first j branches from B, then the starting condition C0(X) is expressed by

$$\underset{i=1}{\overset{j}{AND}}(qi(F(X)) \Rightarrow Ci(F(X))) \qquad (2)$$

Again, some conditions, such as data types and sizes, may be validated at certain points by the underlying physical or virtual machine; these may be explicitly formulated and included in C0(X) where necessary.

8.3 CONDITION DERIVATION ACROSS LOOPS

Because of the possible presence of loops in S, conditions may be derived for branches from B to which conditions have already been attached. To avoid nonterminating iterations of this type, it is necessary to determine an "invariant," a predicate expressing those conditions on the variables involved, that remain invariant during execution of the loop. In the example illustrated in Figure 4, by starting with the single final condition $x < 100$ and mechanically deriving expressions for C0 and C2 in turn, using the formulas

$$C0 ::= \{ [i=10 \Rightarrow C1(x+a(10))] \text{ AND } [i<10 \Rightarrow C2(x+a(i))] \}$$

and

$$C2 ::= C0(i+1)$$

(instances of formulas (2) and (1) above) it is easily seen that C0 generalizes to

$$x + \sum_{j=i}^{10} a[j] < 100 ,$$

from which the derivation back through B0 leads to the initial condition

$$\sum_{j=1}^{10} a[j] < 100 .$$

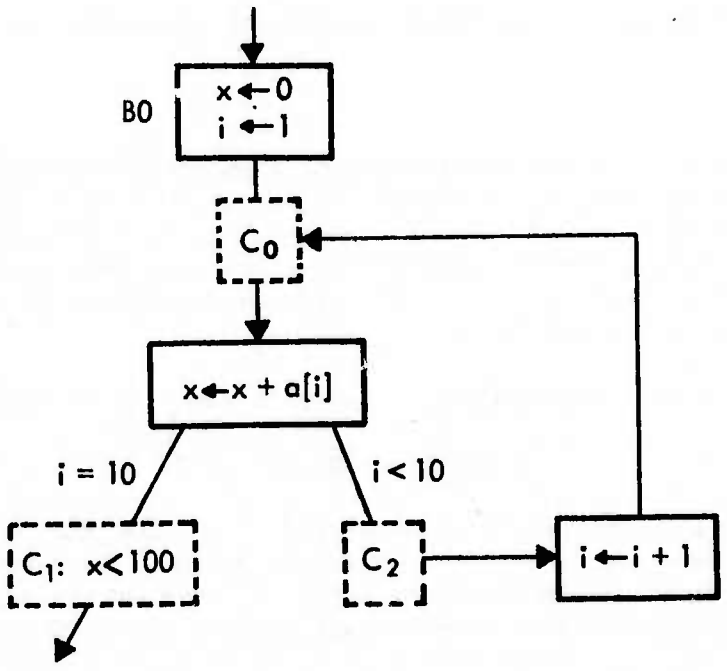

Figure 4. Primitive loop example

In this example, the invariant condition could also have been readily arrived at by mental deduction. If all basic operations and loops were this tractable, derivation of input conditions for a procedure could be largely automated. Many of the necessary techniques and programs are already available from the area of program verification, such as the generation of conditions across certain types of basic operations (e.g., those expressed as algebraic functions). Basic verification methods are introduced in [London75] and [Elspas72], and the state of the art of generating invariances is indicated by [German75] and [Katz76]. Many types of operations occur in operating systems, such as allocation and deallocation, composition and decomposition, insertion and deletion, searching, and reordering--typically involving complex data structures--for which the derivation of conditions may require sophisticated heuristic techniques and considerable intellectual effort. Except for validations known to be well localized, the derivation activity is best regarded as one of man-machine interaction, with the machine performing the more methodical tasks of logical analysis and expression manipulation.

8.4 TERMINATION AND CONTINUATION CONSIDERATIONS

Having derived initial conditions C0(X) for a basic block B, several contingencies exist. If C0(X) is a tautology (true regardless of the values of X), then no conditions are implied for antecedent blocks, and the validation has been shown to be sufficient along control paths to v from B. If the opposite is true, and C0 is logically inconsistent (identically false), then an apparent validation error has been found: Cv(X) cannot be satisfied along the given control paths. This is the "functional" type of error referred to in Section 6.4. In either case, the derivation along paths leading to B is terminated (in the latter case, under the assumption that control can actually reach B). Sometimes an apparent error may be discovered, on closer analysis, to be symptomatic of an improper formulation of Cv(X) in the first place, requiring modification and reevaluation.

If C0(X) is meaningful (neither tautological nor inconsistent) and the entry point of B represents an input point u of some element x of X, then if x is directly modifiable by user-written procedures (as when the entry point of B is a user-callable entry point and x is a parameter), a different type of validation error has been found: a user-modifiable variable has been identified for which certain values can cause Cv(X) not to be satisfied. This is the "exploitable" type of error referred to in Section 6.4.

If x is not directly user-modifiable, then for each output point w of x, a critical item [w, Cw(x,X)] must be joined to the set of critical items awaiting processing, where Cw(x,X) consists of the conjunction of those terms of C0(X) involving x. If any of these terms also involves variables local to the given procedure, then a functional error has been detected, since such terms can be enforced by neither the given procedure nor by procedures modifying x.

Because of the possible presence of cycles in the communication graph, there is no guarantee that the evaluation with respect to [v,Cv(X)] will terminate "naturally" along every control path leading to v, eventually deriving tautological or inconsistent conditions or encountering user-modifiable variables. The same output point w can be specified in more than one critical item generated during the course of the evaluation, and thus derivation of conditions across the section of procedure terminating at w can occur repeatedly. This is analogous to the problem associated with loops within procedures, and in principle can be treated the same way. If an invariance cannot be readily determined or, more generally, if the search has proceeded for what seems to be an unreasonable number of steps up a data path or the evaluation along a data path

otherwise appears inconclusive, it may be truncated. Such an outcome suggests that implicit validation is being depended on to a high degree, indicating the likely existence of subtle validation errors, and the appropriateness of including additional explicit validation along the given path.

REFERENCES

[Allen70] Allen, Frances E., "Control Flow Analysis," *SIGPLAN Notices*, Vol. 5, No. 7, July 1970, pp. 1-19.

[Allen74] Allen, F.E., "Interprocedural Data Flow Analysis," *Information Processing 74*, North-Holland Publishing Company, 1974, pp. 398-402.

[Allen76] Allen, F.E., and J. Cocke, "A Program Data Flow Analysis Procedure," *Communications of the ACM*, Vol. 19, No. 3, March 1976, pp. 137-147.

[Bisbey75] Bisbey, Richard, Gerald Popek, and Jim Carlstedt, *Protection Errors in Operating Systems: Inconsistency of a Single Data Value Over Time*, USC/Information Sciences Institute, ISI/SR-75-4, December 1975.

[Bisbey76] Bisbey, Richard, Jim Carlstedt, Dale Chase, and Dennis Hollingworth, *Data Dependency Analysis*, USC/Information Sciences Institute, ISI/RR-76-45, February 1976.

[Carlstedt75] Carlstedt, Jim, Richard Bisbey, and Gerald Popek, *Pattern-Directed Protection Evaluation*, USC/Information Sciences Institute, ISI/RR-75-31, June 1975.

[Denning75] Denning, Dorothy E., *Secure Information Flow in Computer Systems*, Purdue University, Department of Computer Sciences, TR-145, May 1975.

[Elspas72] Elspas, B., K.N. Levitt, and R.J. Waldinger, "An Assessment of Techniques for Proving Program Correctness," *Computing Surveys*, Vol. 4, No. 2, June 1972, pp. 97-147.

[Ford62] Ford, L.R., Jr., and Fulkerson, D.R., *Flows in Networks*, Princeton University Press, 1962.

[German75] German, Steven M., and Ben Wegbreit, "A Synthesizer of Inductive Assertions," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975, pp. 68-75.

[Goodenough75] Goodenough, John B., "Exception Handling: Issues and a Proposed Notation," *Communications of the ACM*, Vol. 18, No. 12, December 1975, pp. 683-696.

[Jones73] Jones, Anita Katherine, *Protection in Programmed Systems*, Carnegie-Mellon University, Department of Computer Science, June 1973.

[Katz76] Katz, Shmuel, and Zohar Manna, "Logical Analysis of Programs," *Communications of the ACM*, Vol. 19, No. 4, April 1976, pp. 188-206.

[Kildall73] Kildall, Gary A., "A Unified Approach to Global Program Optimization," *Conference Record of ACM Symposium on Principles of Programming Languages*, October 1973, pp. 194-206.

[Lampson69] Lampson, B. W., "Dynamic Protection Structures," *AFIPS Proceedings, Fall Joint Computer Conference*, 1969, pp. 27-38.

[Lampson73] Lampson, B. W., "A Note on the Confinement Problem," *Communications of the ACM*, Vol. 16, No. 10, October 1973, pp. 613-615.

[Levin75] Levin, R., E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism Separation in Hydra," *Operating System Review*, Vol. 9, No. 5, 1975, pp. 132-140.

[London75] London, Ralph L., "A View of Program Verification," *SIGPLAN Notices*, Vol. 10, No. 6, June 1975, pp. 534-545.

[McPhee74] McPhee, W.S., "Operating System Integrity in OS/VS2," *IBM Systems Journal*, Vol. 13, No. 3, 1974, pp. 230-252.

[Saltzer75] Saltzer, Jerome H., and Michael D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, Vol. 63, No. 9, September 1975, pp. 1278-1308.

[Schroeder72] Schroeder, Michael D., and Jerome H. Saltzer, "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM*, Vol. 15, No. 3, March 1972, pp. 157-170.